



Алгоритмы и структуры данных

Лекция 7. Стек и очередь на массиве

Антон Штанюк (к.т.н, доцент)

31 марта 2022 г.

Нижегородский государственный технический университет им. Р.Е. Алексеева
Институт радиоэлектроники информационных технологий
Кафедра "Компьютерные технологии в проектировании и производстве"

Определение стека

Реализация стека на статическом массиве

Реализация стека на динамическом массиве

Реализация стека на шаблоне класса

Определение очереди

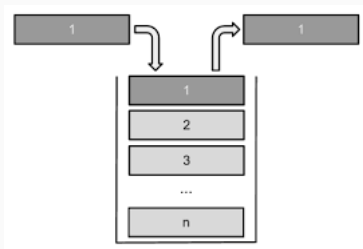
Программная реализация очереди

Список литературы

В этой и следующей темах мы остановимся на популярных абстрактных структурах данных - **стеке** и **очереди**. Абстрактность здесь обозначает то, что работа СД определяется не способом построения, а только принципом доступа к элементам. В дальнейшем мы рассмотрим примеры реализации на массивах и на списках.

Определение стека

Структура данных **Стек** является широко известной и очень распространенной. В ее основе лежит принцип доступа к элементам по названию **LIFO** - последний пришел, первый вышел.



Элементы помещаются и извлекаются с головы стека (**top**)

Реализация на массиве предполагает выбор типа массива:

- На массиве фиксированной длины (статическом массиве)
- На массиве переменной длины (динамическом массиве)

Добавление элемента и его удаление выполняются за время $O(1)$

Реализация стека на статическом массиве

Рассмотрим реализацию стека на статическом массиве. Реализацию сложных СД на объектно-ориентированных языках программирования лучше приводить в виде классов.

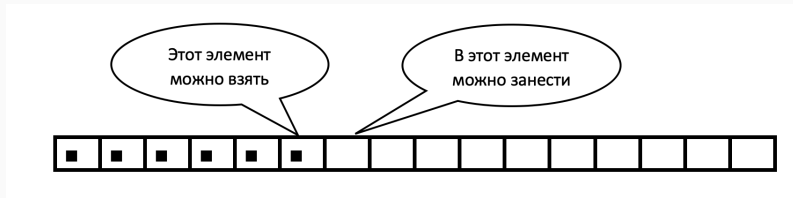
```
typedef type int;
const int size=100;

class SStack {
    private:
        type arr[size];
        int top;
    public:
        SStack():top(-1) { }
        type get() const {
            return arr[top];
        }
    ...
}
```



```
...  
    bool isEmpty() const {  
        return top==-1;  
    }  
    bool isFull() const {  
        return top==size-1;  
    }  
    void pop() {  
        if(top>=0)  
            top--;  
    }  
    void push(type item) {  
        if(top<size-1)  
            arr[++top]=item;  
    }  
};
```

- Данные хранятся в массиве **arr** в закрытой области класса (доступ напрямую в эту область запрещен).
- Для доступа к данным мы используем функции **get, push, pop**, которые позволяют контролировать доступ.
- Операция **get** возвращает элемент на вершине стека.
- Операция **pop** извлекает последний элемент.
- Операция **push** добавляет новый элемент в стек.
- Функции **isEmpty, isFull** проверяют, пустой ли стек или полный.
- Переменная класса **top** указывает на границу данных в массиве.
- При опустошении и переполнении стека ничего не происходит, операции игнорируются.
- Стек можно настроить для хранения данных любого типа, нужно отредактировать строку с **typedef**.



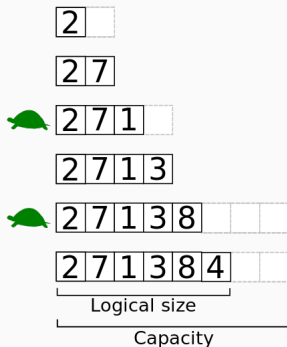
```
...  
SStack stack; // создание нового стека  
stack.push(5); // добавление первого элемента  
stack.push(7); // добавление второго элемента  
stack.push(11); // добавление третьего элемента  
  
std::cout<<stack.get(); // вывод 11  
stack.pop();  
std::cout<<stack.get(); // вывод 7  
stack.pop();  
std::cout<<stack.get(); // вывод 5  
stack.pop();  
...
```

Можно написать реализацию стека, в которой метод **pop** возвращает элемент. Проблема лишь в том, что в случае, когда стек пуст, неясно, что нужно возвращать. Вариантов решения несколько:

- Возвращать **true, false** как результат любой операции, а данные записывать по указателю (ссылке).
- Выбрасывать исключение оператором **throw**, если стек пуст или полон.

Реализация стека на динамическом массиве

Реализация стека на динамическом массиве предполагает, что если исходный массив будет заполнен, стек автоматически увеличит свою память чтобы вместить новые элементы.



Увеличение размера массива происходит быстро пока он меньше объёма массива. Когда нужно увеличить размер массива, а свободного места в нём нет, создаётся ещё один массив большего объёма, все элементы старого объёма копируются в новый массив, ссылка на старый массив удаляется.


```
typedef type int;

class DStack {
private:
    type *arr;
    int size;
    int top;
    void resize(int nsize) {
        type *temp=new type[nsize];
        for(int i=0;i<size;i++)
            temp[i]=arr[i];
        delete[]arr;
        arr=temp;
        size=nsize;
    }
public:
    DStack(int size):top(-1) {
        this->size=size;
        arr=new type[size];
    }
    ...
};
```

```
...  
~DSize() {  
    delete[] arr;  
}  
bool isEmpty() const {  
    return top==−1;  
}  
type get() const {  
    return arr[top];  
}  
void pop() {  
    if(top>=0)  
        top−−;  
}  
void push(type item) {  
    if(top==size−1)  
        resize(2*size);  
    arr[++top]=item;  
}  
};
```

- При создании стека нужно указать первоначальный размер массива.
- В классе **DStack** определяется закрытый метод **resize** для увеличения размера памяти.
- Поскольку при использовании **DStack** выделяется динамическая память, то в классе определяется специальный метод - *деструктор DStack*, который освобождает выделенную в конструкторе динамическую память.
- Используемая память может только увеличиваться, освобождение происходит при разрушении стека.

Реализация стека на шаблоне класса

Если в одной программе нужно использовать несколько стеков с различными типами данных, то лучшее решение будет состоять в использовании механизма шаблонов.

```
const int size=100;

template<typename type>
class TStack {
    private:
        type arr[size];
        int top;
    public:
        TStack():top(-1) { }
        type get() const {
            return arr[top];
        }
    ...
}
```

```
...  
    bool isEmpty() const {  
        return top==-1;  
    }  
    bool isFull() const {  
        return top==size-1;  
    }  
    void pop() {  
        if(top>=0)  
            top--;  
    }  
    void push(type item) {  
        if(top<size-1)  
            arr[++top]=item;  
    }  
};
```

Рассмотрим теперь, как можно воспользоваться таким стеком:

```
#include "tstack.h"
#include <iostream>

int main() {
    TStack<int> istack;
    TStack<char> cstack;

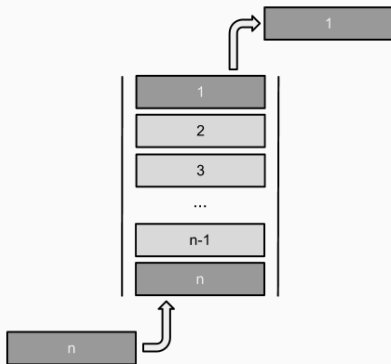
    for(int i=1; i<=10; i++)
        istack.push(i);

    while(istack.isEmpty()==false) {
        std::cout<<istack.get()<<" ";
        istack.pop();
    }
    std::cout<<std::endl;
    return 0;
}
```

Определение очереди

Определение очереди

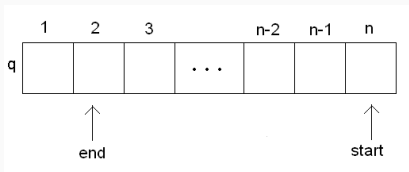
Структура данных **Очередь**, также как и стек, является широко распространенной и популярной. В ее основе лежит принцип доступа к элементам под названием **FIFO - первый пришел, первый вышел**.



Реализация очереди на массиве предполагает тоже два подхода:

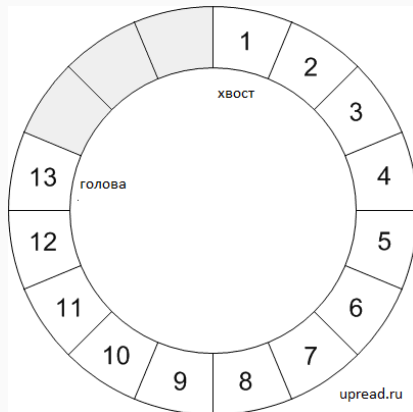
- на основе массива фиксированной длины (статического)
- на основе массива переменной длины (динамического)

Реализация на массиве предполагает использование двух указателей: на начало очереди и на ее конец. В процессе работы оба указателя движутся в одно направлении и неизбежно упрутся в правую границу массива.



Определение очереди

Данное обстоятельство приводит к использованию массива фиксированной длины, но с возможностью перехода на начальные элементы массива. Возникает эффект кольца, поэтому такую реализацию часто называют **очередью на кольцевом буфере**.



Программная реализация очереди

Реализуем очередь в виде шаблонного класса с использованием массива фиксированного размера. Описание шаблонного класса мы разобьем на две части: сам шаблонный класс и описание его методов.

```
#include <cassert>

template<typename T>
class TQueue {
private:
    T *arr;           // массив с данными
    int size;         // максимальное количество элементов в очереди размер(
                      // массива)
    int begin,        // начало очереди
        end;          // конец очереди
    int count;        // счетчик элементов
public:
    TQueue(int =100); // конструктор по умолчанию
    ~TQueue();        // деструктор
    ...
};
```

```
...  
    void push(const T &); // добавить элемент в очередь  
    T pop();              // удалить элемент из очереди  
    T get() const;        // прочитать первый элемент  
    bool isEmpty() const; // пустая ли очередь?  
    bool isFull() const;  // заполнен ли массив?  
};
```

Далее, рассмотрим описание конструктора, который будет автоматически вызываться при создании очереди. На вход конструктора подается размер очереди в элементах.

```
// конструктор по умолчанию
template<typename T>
TQueue<T>::TQueue(int sizeQueue) :
    size(sizeQueue),
    begin(0), end(0), count(0) {
    // дополнительный элемент поможет нам различать конец и начало очереди
    arr = new T[size + 1];
}
```

Деструктор будет автоматически вызываться при уничтожении очереди и его главная задача - освободить память, выделенную под массив.

```
// деструктор класса Queue  
template<typename T>  
TQueue<T>::~~TQueue() {  
    delete [] arr;  
}
```


Функция добавления элемента в очередь с проверкой необходимости перехода в начало массива.

```
template<typename T>
void TQueue<T>::push(const T & item) {
    // проверяем, есть ли свободное место в очереди
    assert( count < size );

    arr[end++] = item;
    count++;

    // проверка кругового заполнения очереди
    if (end > size)
        end -= size + 1; // возвращаем end на начало очереди
}
```

Функция удаления элемента из очереди с проверкой необходимости перехода в начало массива.

```
// функция удаления элемента из очереди
template<typename T>
T TQueue<T>::pop() {
    // проверяем, есть ли в очереди элементы
    assert( count > 0 );

    T item = arr[begin++];
    count--;

    // проверка кругового заполнения очереди
    if (begin > size)
        begin -= size + 1; // возвращаем begin на начало очереди

    return item;
}
```

Функция чтения элемента из начала очереди, не меняя состояние очереди.

```
// функция чтения элемента на первой позиции
template<typename T>
T TQueue<T>::get() const {
    // проверяем, есть ли в очереди элементы
    assert( count > 0 );
    return arr[begin];
}
```

```
// функция проверки очереди на пустоту
template<typename T>
bool TQueue<T>::isEmpty() const {
    return count==0;
}
```

```
// функция проверки очереди на заполненность
template<typename T>
bool TQueue<T>::isFull() const {
    return count==size;
}
```

Рассмотрим детали реализации очереди:






- Переменная **size** хранит максимальную емкость кольцевого буфера.
- Переменная **count** хранит реальное количество элементов в очереди.
- Конструкция **assert** проверяет результат выражения и если он **false** аварийно завершает программу.
- При достижении указателями **begin**, **end** правой границы массива, они переводятся на левую границу.






В следующей программе мы создаем очередь из целых чисел с максимальным размером 50 элементов, заполняем ее числами от 1 до 45, а потом выводим значения на экран.

```
#include "tqueue.h"
#include <iostream>

int main() {
    TQueue<int> que(50);
    for(int i=1; i<=45; i++)
        que.push(i);
    while(que.isEmpty()==false)
        std::cout<<que.pop()<<std::endl;
    return 0;
}
```

Список литературы

-  Кормен Т., Лейзерсон Ч., Ривест Р.
Алгоритмы: построение и анализ
МЦНМО, Москва, 2000
-  Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы:
построение и анализ.
2-е изд. — М.: «Вильямс», 2006
-  Википедия
Алгоритм
<http://ru.wikipedia.org/wiki/Алгоритм>
-  Википедия
Список алгоритмов
http://ru.wikipedia.org/wiki/Список_алгоритмов
-  Традиция
Задача коммивояжёра
<http://traditio.ru/wiki/Задача>

-  Википедия
NP-полная задача
<http://ru.wikipedia.org/wiki/NP-полная>
-  Серджвик Р.
Фундаментальные алгоритмы на C++. Части 1-4
Diasoft, 2001
-  Седжвик Р.
Фундаментальные алгоритмы на C. Анализ/Структуры данных/Сортировка/Поиск
СПб.: ДиаСофтЮП, 2003
-  Седжвик Р.
Фундаментальные алгоритмы на C. Алгоритмы на графах
СПб.: ДиаСофтЮП, 2003
-  Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы.
Издательский дом «Вильямс», 2000



Кнут Д.

Искусство программирования, том 1. Основные алгоритмы
3-е изд. — М.: «Вильямс», 2006



Кнут Д.

Искусство программирования, том 2. Получисленные методы
3-е изд. — М.: «Вильямс», 2007



Кнут Д.

Искусство программирования, том 3. Сортировка и поиск
2-е изд. — М.: «Вильямс», 2007



Кнут Д.

Искусство программирования, том 4, выпуск 3. Генерация всех сочетаний и разбиений
М.: «Вильямс», 2007



Кнут Д.

Искусство программирования, том 4, выпуск 4. Генерация всех деревьев. История комбинаторной генерации

М.: «Вильямс», 2007